HPC, CUDA task report

Jan Kwiatkowski

10.05.2024

Overall solution structure

I've implemented 4 different solutions. In this section I will present the main ideas which all of them have in common. Steps of the solution:

- Indexing.
  All fragments of code responsible for database indexing are present in *common.cpp* file.

  1. Read the database in parts - it is to large to read it whole at once into the program's memory.
  2. Compute some hashes described below.
  3. After computing hashes save them with the original line from the database into index file.

  Hashing is the main idea behind my solution. Firstly notice that each line can be **uniquely** represented by some big number - we can concatenate columns (chrom, pos, ref, alt) and store them in one 64-bit number. Since in the task we assume that such tuple uniquely determines the line from the database, such a representation is unique.

  However this hash is not enough. If we were working on CPU only it would not be a problem - we would store the hashes and lines in a structure like *unordered_map*. We cannot send such structure to GPU, therefore we compute the second, much smaller hash, which can be used as an array index. This hash is not bigger than $MAX\_HASH = 1024 \cdot 1024$.

  Of course some collisions may occur. In my solution I assume (based on the provided example file and my hash function) that no more than $MAX\_COLLISIONS = 10$ collisions per hash will occur.

  Because the database is big we have to divide its content into chunks and process them in chunks. I've chosen $CHUNK\_SIZE = 512 \cdot 1024$. Small hash can be twice as big, so we avoid many collisions that way.

  To summarise - for each line in database I compute it's unique representation and it's small hash and save them together with the line in index file.

- Processing variant file.
  Implementation differs between my solutions but the main idea is as follows:

  1. Read the variant file.
  2. Proceed through index chunks.
  3. For each chunk store all small hashes and unique representations in an array.
  4. Proceed through variant file and for each of its lines check if some representation matches with the one from index file.

  That means that we iterate over the whole variant file for each database chunk.

Implemented solutions

1. *cpugenv.cpp*
   Whole computation is done on cpu. Therefore the code is short and pretty clean. If you plan to look at other implementations, I strongly recommend reading this code first. It shows all main ideas which will be implemented later.

2. *gpugenv.cu*
   Basic implementation on the GPU. Firstly we save the hashes and representations of lines from the variant file and copy them to GPU. Then we do the same thing as in the previous solution. The main difference is that we divide the work - each thread will check a small part of the variant file. Of course such computation will happen once per each chunk.

   The easiest way to implement it would require sending the strings (whole lines from the database) to GPU, however copying the strings to GPU seems like a pretty bad idea. Therefore the code gets more complicated - instead of sending hashmap (hash, whole line) we send the hashmap (hash, unique representation) to GPU. Then GPU for each variant line saves the index in the hashtable, so we can obtain a true result (not only a unique representation) back on CPU.

   This is the main cause why the GPU codes are more complicated than the CPU code.

3. *gpugenv_continuous.cu*
   If you take a look at the implementation of the GPU kernel in the previous file you will notice that it jumps through the *variant* arrays:

   ```
   for (size_t i = start_idx + threadIdx.x; i < end_idx; i += blockDim.x)
   ```

   In this version of the code each thread will work with continuous fragment of the *variant* arrays.

   ```
   for (size_t i = start_delta; i < end_delta; i++)
   ```

4. *gpugenv_stream.cu*
   The logic of the code remains the same, but code itself gets much more complicated due to boilerplate needed while using threads (page-locked memory etc.). We use streams, so that, at once:

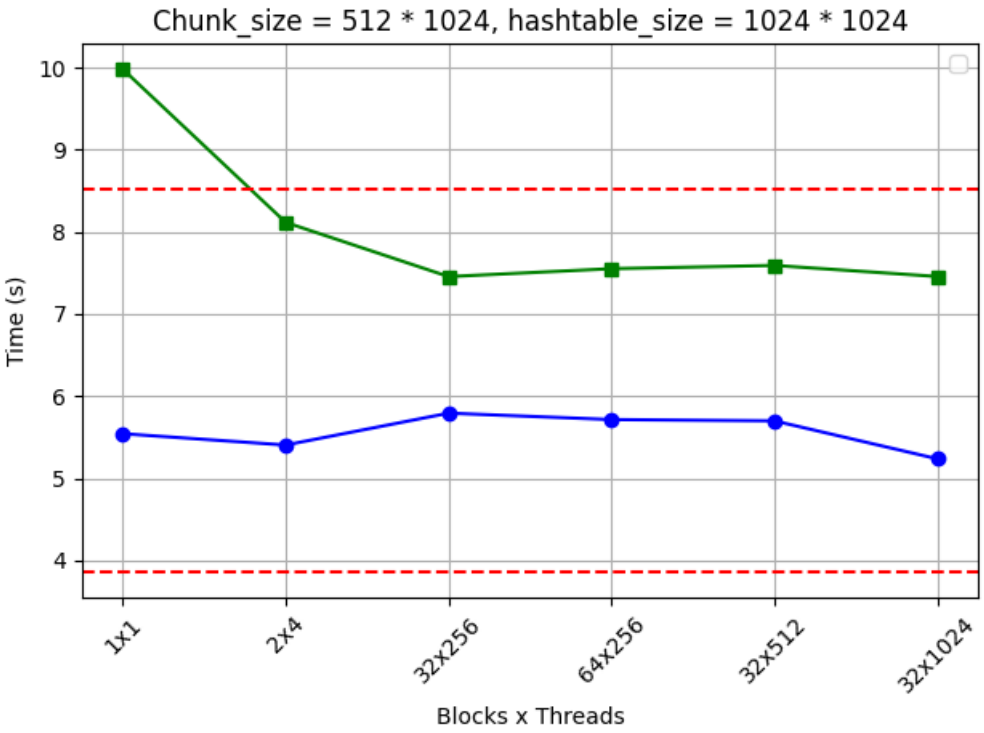
   - GPU computes the results;
   - CPU reads the content of index file.

   So this optimization allows two different things in the code to happen at once.

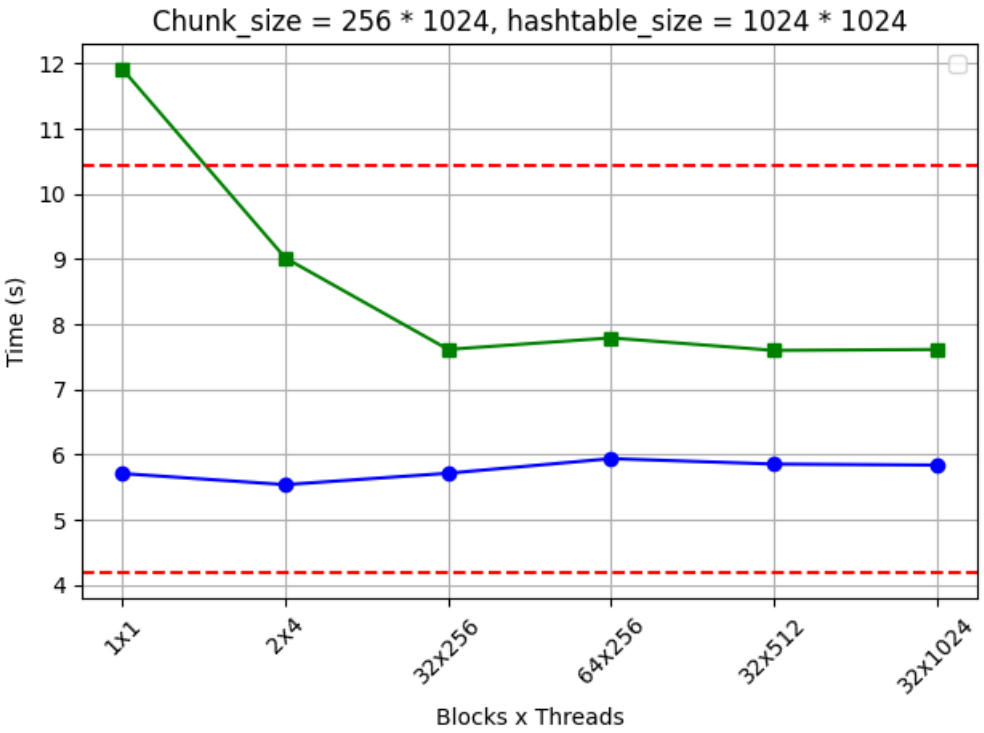
Benchmarks

Solutions *gpugenv.cu* and *gpugenv_continuous.cu* were indistinguishable, their times were nearly identical. Solution *gpugenv_streams.cu* was usually around 0,3s slower than the previous two with one small exception: it was visibly (around 1,5s faster) when testing on 1 block and 1 thread per block. That makes sense since 1 thread, 1 block means no parallelism at all, so solution with streams had a handicap of asynchronous operations. In other cases it was probably slower because of stream overhead. If the kernel function usage was higher, then probably stream solution would be faster.

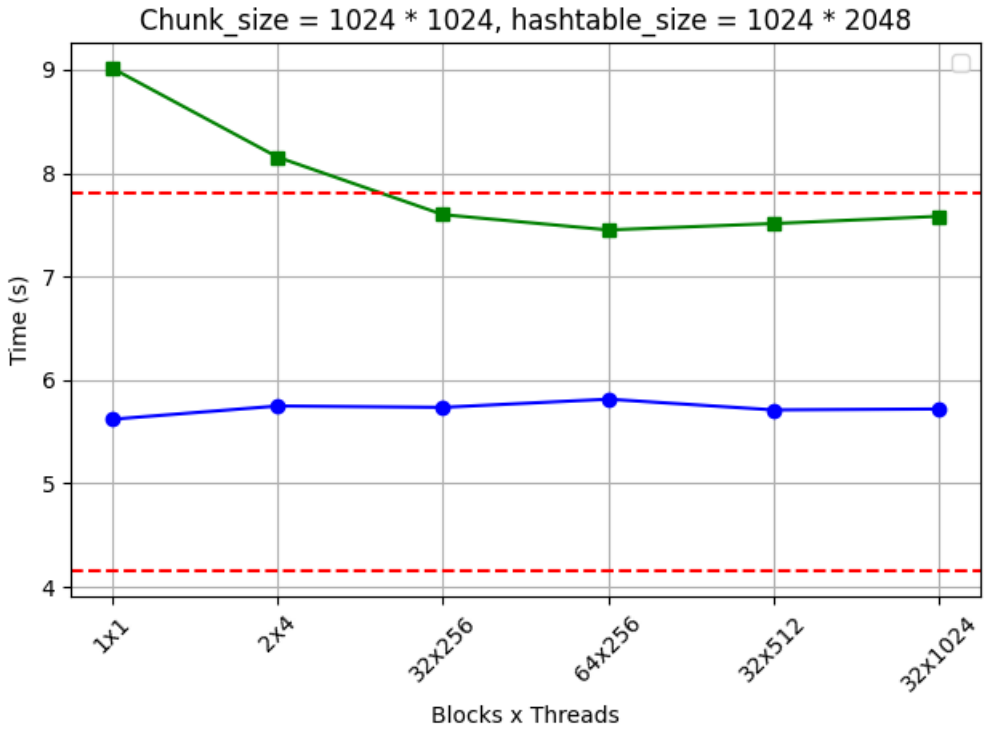
Here are the charts with some of the benchmarks I've conducted. Blue line presents the average runtime of *gpugenv.cu* on example variant file. Green line present the average runtime of *gpugenv.cu* on variant file which contains around 1'000'000 lines. With red-dotted line I've marked a CPU solution times.



Drawing 1: The first benchmark.



Drawing 2: The second benchmark.



Drawing 3: The third benchmark.

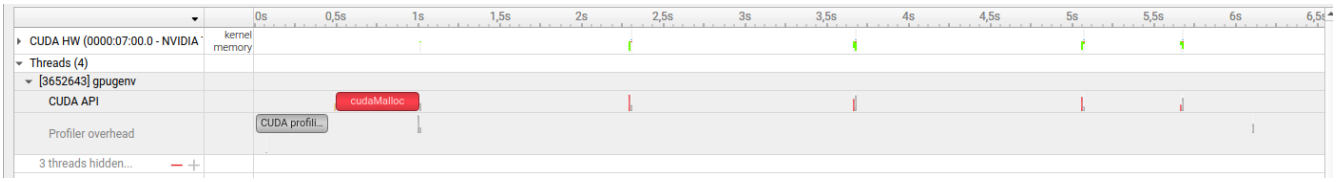
To conclude, the GPU solution was visibly faster, but only on large enough variant files. Runtimes on bigger number on blocks and threads didn't differ too much from each other, however one curious observation is that the less blocks and threads I've tried, the bigger the variance in runtime was. Maybe it means that more blocks, threads result in more stability.

The optimal size of the hashtable seems to be around 2 times bigger than the size of the chunk. Also as can be seen if the hashtable is too big then the GPU solution is only a tiny bit better than CPU solution.

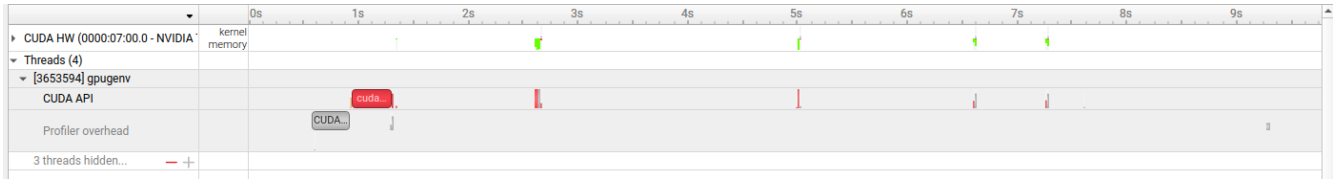
From the benchmarks I've chosen some final constants which can be found in *gpugenv.cu*.

NSIGHT Systems

Here's some valuable information gathered via the profiler. Most images will come in two version: one for the example test from the task and one for a test where *variant.vnf* has around 1'000'000 lines. Let's start with looking at the *gpugenv.cu*:



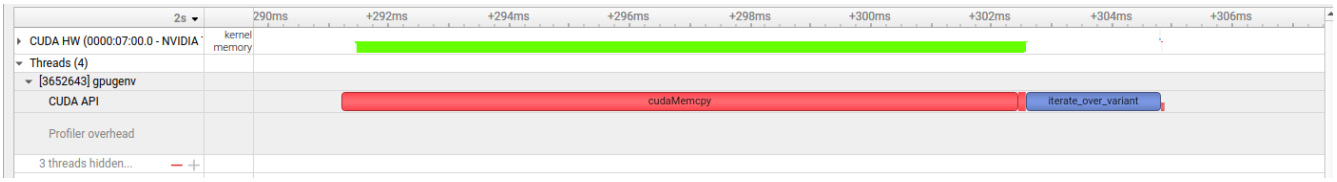
Drawing 4: Example test visualization.



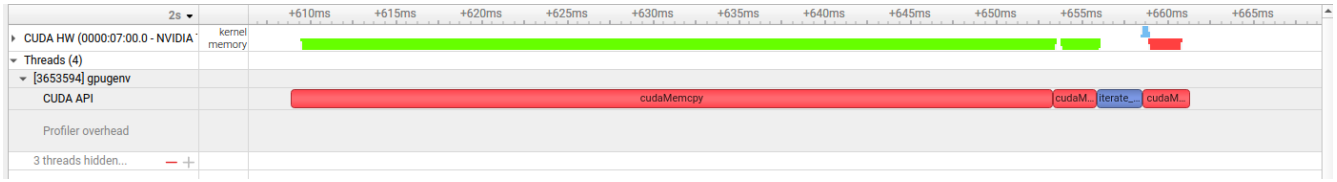
Drawing 5: Big test visualization.

We can see that the vast majority of our program's work is done outside of GPU - reading the index from disk is much more costly.

Here we can see how the kernel behave:



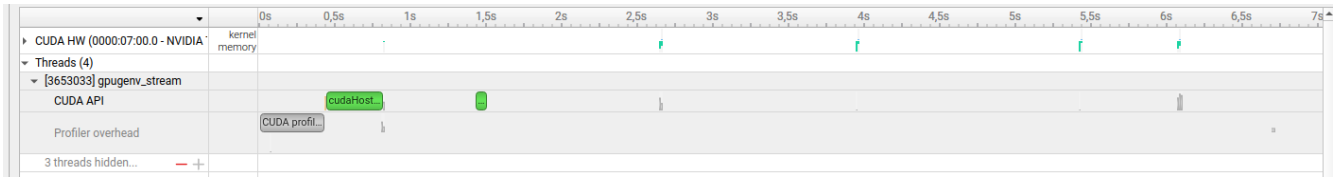
Drawing 6: Example test visualization.



Drawing 7: Big test visualization.

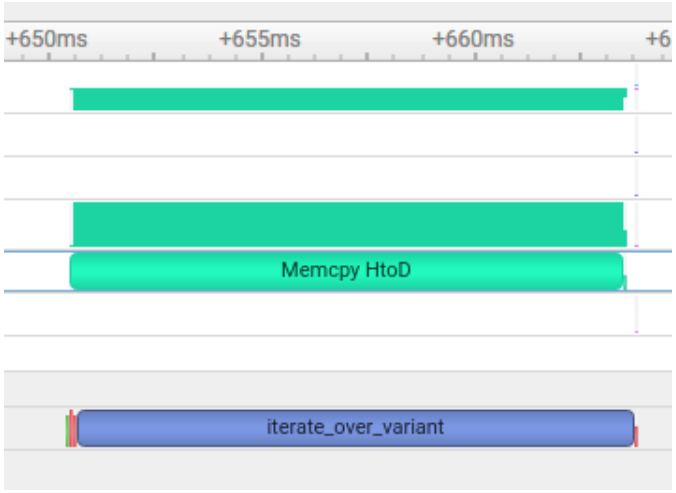
We can reason that there is not that much to do on GPU since Memcpy takes longer than the proper kernel execution. Also we can see that the proportion between Memcpy and kernel length grows if size of the variant file grows.

Here we can see how the *gpugenv_stream.cu* solution behaves:



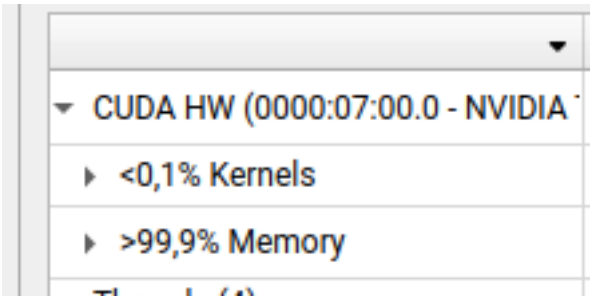
Drawing 8: Example test stream visualization.

We notice that indeed there is no cudaMalloc but there is cudaHostAlloc. We can also notice that streams indeed parallelize CPU and GPU work:

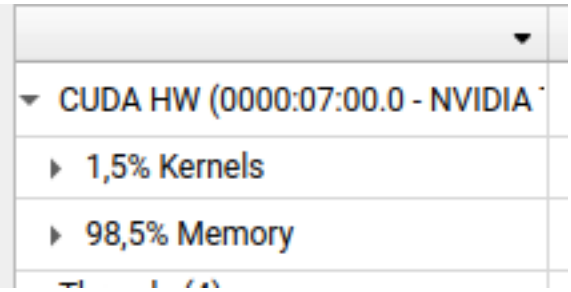


Drawing 9: Stream parallelism visualization.

In the end the conclusions are simple - it is hard to obtain big speedups since kernel work is only a small fraction of the whole work.



Drawing 10: Example test kernel percent.



Drawing 11: Big test kernel percent.